Automatisierte Logik und Programmierung

Einheit 17

Anwendungen formaler Systeme

1. **Mathematik:**
   Automating Proofs in Category Theory
2. **Programmierung:**
   Automated Fast-Track Reconfiguration
   of Group Communication Systems

Automating Proofs in Category Theory

Christoph Kreitz

Institut für Informatik, Universität Potsdam
&
Department of Computer Science, Cornell University

1. Kozen's proof calculus
2. Implementation in Nuprl
3. Proof automation
4. Insights, future questions

Joint work with Dexter Kozen (Cornell University) and Eva Richter (Universität Potsdam)

Presented at "Logics and Program Semantics", Cornell University, April 2012

WHY CATEGORY THEORY?

- **Example: Currying in Programming Languages**
  - The types $C \times D \to E$ and $C \to D \to E$ are considered isomorphic
  - Rationale: transform $f : C \times D \to E$ into $\lambda x.\lambda y.f(x, y) : C \to D \to E$
    This is clearly a bijection

Why category theory?

- **Example: Currying in Programming Languages**
  - The types $C \times D \to E$ and $C \to D \to E$ are considered isomorphic
  - Rationale: transform $f : C \times D \to E$ into $\lambda x.\lambda y.f(x,y) : C \to D \to E$
    This is clearly a bijection

  But isomorphisms are more than just bijections
  - An isomorphism ι between types A and B has to preserve structure
    - · ι is a bijection between objects in A and those in B
    - · ι transforms operations on A into operations on B
      such that $g(\iota(x)) = \iota(f(x))$ whenever $\iota(f) = g$
  - Isomorphisms can also be defined for sets, groups, automata, ...

WHY CATEGORY THEORY?

- **Example: Currying in Programming Languages**
  – The types $C{\times}D{\rightarrow}E$ and $C{\rightarrow}D{\rightarrow}E$ are considered isomorphic
  – Rationale: transform $f : C{\times}D{\rightarrow}E$ into $\lambda x.\lambda y.f(x,y) : C{\rightarrow}D{\rightarrow}E$
  This is clearly a bijection

  But isomorphisms are more than just bijections
  – An isomorphism ι between types A and B has to preserve structure
  · ι is a bijection between objects in A and those in B
  · ι transforms operations on A into operations on B
  such that $g(\iota(x)) = \iota(f(x))$ whenever $\iota(f) = g$
  – Isomorphisms can also be defined for sets, groups, automata, ...

- **Category Theory analyzes common structures**
  What properties of mathematical domains depend only on structure?
  · Focus on mathematical objects and “morphisms” on these objects
  · Develop a generic framework for expressing abstract properties
  – Results have a wide impact on mathematics and computer science

Proving Properties in Category Theory

- **Category theory is an elegant formalism**
  - Framework for expressing properties common to set theory, logic, algebra, topology, semantics, software specification, etc.
  - High level of abstraction makes constructions elegant and precise
  - Diagrams provide key insights and proof ideas
- **Most proofs are considered straightforward**
  - Arguments are based on standard patterns of reasoning
  - Many steps of a detailed proof rely on pure symbol manipulation
- **Detailed proofs contain too many steps**
  - Even basic proofs can be very tedious
  - Humans cannot verify the details of a proof
  - How can we be sure they are correct?

Can category theoretical proofs be automated?

… THIS IS WHERE THINGS GOT STARTED

DK: “*Do you know an automated proof system for Category Theory?*”

CK: “*No … but it shouldn’t be too difficult to build one*”

Formal proofs in category theory

- **Formalized category theory** (Mizar project, 1990-2001)
  - Focus of formalization of mathematical knowledge
  - Automated proof checking but no support for proof search
- **Embedding into logical calculi** (Isabelle/HOL, Coq, 1996—)
  - Focus on development of formal proofs
  - Interactive proof search with some tactical support
  - Formalization strongly depends on theory underlying the proof system
- **Our goal:**
  - It should be possible to formalize reasoning patterns as proof rules
    - Use an independent, simple "first-order" calculus (Kozen, 2004)
    - Faithfully implement calculus in theorem proving environment
  - It should be possible to completely automate "trivial" proofs since key insights are often considered "the only obvious choice"

Fundamentals of Category Theory ... informally

- **A category $\mathcal{C}$ consists of**
  - A class obj($\mathcal{C}$) of objects, briefly denoted by C
  - A class hom($\mathcal{C}$) of morphisms ("arrows"), where each morphism f has a domain $A : \mathsf{C}$ and a codomain $B : \mathsf{C}$, denoted by $f : \mathsf{C}(A, B)$
  - A binary composition of morphisms, denoted by $g \circ f$, where $g \circ f : \mathsf{C}(A, D)$ if $f : \mathsf{C}(A, B)$ and $f : \mathsf{C}(D, B)$
    $\circ$ must be associative, i.e. $h \circ (g \circ f) = (h \circ g) \circ f$ for all f, g, h
  - An identity $1_A : \mathsf{C}(A, A)$ for each $A : \mathsf{C}$ such that $f \circ 1_A = f = 1_B \circ f$ for all $f : \mathsf{C}(A, B)$
- **A Functor F : $\mathsf{Fun}\,[\mathsf{C}, \mathsf{D}]$ consists of**
  - A mapping on objects $F^1 : \mathsf{C} \rightarrow \mathsf{D}$
  - A mapping on morphisms $F^2 : \mathsf{C}(A, B) \rightarrow \mathsf{D}(F^1A, F^1B)$ such that $F^2(1_A) = 1_{F^1A}$ and $F^2(g \circ f) = F^2g \circ F^2f$ for all A, f, g
- **Natural Transformation φ : $\mathsf{Fun}\,[\mathsf{C}, \mathsf{D}]\,(\mathsf{F}, \mathsf{G})$**
  - Mapping between functos $F, G : \mathsf{Fun}\,[\mathsf{C}, \mathsf{D}]$ such that $\varphi B \circ F^2 g = G^2 g \circ \varphi A$ for all $A, B : \mathsf{C}$, $\mathsf{g} : \mathsf{C}(\mathsf{A}, \mathsf{B})$

Kozen's calculus for elementary category theory

- **First-order axiomatization of basic constructs**
  - Objects $A : \mathsf{C}$, morphisms $f : \mathsf{C}(A, B)$, composition $g \circ f$, identities 1_A
  - Functors $F : \mathsf{Fun}[\mathsf{C}, \mathsf{D}]$, natural transformations $\varphi : \mathsf{Fun}[\mathsf{C}, \mathsf{D}](F, G)$
  - Products $\mathsf{C} \times \mathsf{D}$, dual category C^{op}, large categories Cat
- **Rules involve sequents $\Gamma \vdash \alpha$**
  - Γ type environment of objects/morphisms, α type judgement or equation
  - Analysis rules for decomposition of objects
  - Synthesis rules for construction of objects
  - Extensionality rules for functors and natural transformations
  - Equational rules e.g., for identities (essential for most proofs)

$$\frac{\Gamma \vdash A : \mathsf{C}}{\Gamma \vdash 1_A : \mathsf{C}(A, A)} \qquad \frac{\Gamma \vdash A, B : \mathsf{C} \qquad \Gamma \vdash f : \mathsf{C}(A, B)}{\Gamma \vdash f \circ 1_A = f}$$

Synthesis and analysis rules

- **Synthesis of a functor** F : $\mathsf{Fun}\,[\mathsf{C},\mathsf{D}]$
  – Analyze mapping F^1 on objects and F^2 on morphisms

$$\frac{\begin{array}{r}\Gamma,\ A:\mathsf{C}\ \vdash\ F^1A:\mathsf{D}\\ \Gamma,\ A,B:\mathsf{C}, g:\mathsf{C}(A,B)\ \vdash\ F^2g:\mathsf{D}(F^1A,F^1B)\\ \Gamma,\ A,B,C:\mathsf{C}, g:\mathsf{C}(A,B), h:\mathsf{C}(B,C)\ \vdash\ F^2(h\circ g)=F^2h\circ F^2g\\ \Gamma, D:\mathsf{C}\ \vdash\ F^2(1_D)=1_{F^1D}\end{array}}{\Gamma\ \vdash\ F:\mathsf{Fun}\,[\mathsf{C},\mathsf{D}]}$$

- **Functor analysis** (inverses to synthesis rule)

$$\frac{\Gamma\vdash F:\mathsf{Fun}\,[\mathsf{C},\mathsf{D}]\,,\quad \Gamma\vdash \mathsf{A}:\mathsf{C}}{\Gamma\vdash F^1A:\mathsf{D}}$$

$$\frac{\Gamma\vdash F:\mathsf{Fun}\,[\mathsf{C},\mathsf{D}]\,,\quad \Gamma\vdash \mathsf{A},\mathsf{B}:\mathsf{C},\quad \Gamma\vdash \mathsf{f}:\mathsf{C}(\mathsf{A},\mathsf{B})}{\Gamma\vdash F^2f:\mathsf{D}(F^1A,F^1B)}$$

$$\vdots$$

- **Extensionality**

$$\frac{\begin{array}{r}\Gamma\ \vdash\ F,G:\mathsf{Fun}\,[\mathsf{C},\mathsf{D}]\\ \Gamma, A:\mathsf{C}\ \vdash\ F^1A=G^1A\\ \Gamma, A,B:\mathsf{C}, g:\mathsf{C}(A,B)\ \vdash\ F^2g=G^2g\end{array}}{\Gamma\ \vdash\ F=G}$$

Additional rules

- **Laws of composition and identities**
  – Synthesis, associativity, equational rules for identities
- **Rules for natural transformations**
  – Synthesis, analysis, extensionality
- **Definition of products, dual category, large categories**
- **Standard equality reasoning**
  – reflexivity, symmetry, transitivity, congruence

Calculus is complete for basic category theory

– Detailed formal proofs can be generated by hand

– Proof construction error prone and time consuming

13 pages for proof of $\mathsf{Fun}[C \times D, E] \simeq \mathsf{Fun}[C, \mathsf{Fun}[D, E]]$

... and details of equality & first-order reasoning still had to be omitted

– Calculus needs computer support and automated proof search

Implementation platform: the Nuprl system

- **Infrastructure for interactive proof development**
  - Refinement style proof development with top-down sequent rules
  - Proof automation with user-definable proof tactics
- **Support for mathematical knowledge managment**
  - Proof calculus is explicitly represented in system's library
  - Users can add definitions, theorems/proofs, and proof tactics
  - Expert users can add different proof calculi to the system
- **Standard calculus is constructive type theory**
  - Higher-order logic with expressive, open-ended data type system

Implementing vocabulary and rules

- **Represent concepts as abstract definition objects**
  - Abstract terms for categories, functors, etc. are added to the library
  - Display forms provide a “natural presentation” on the screen

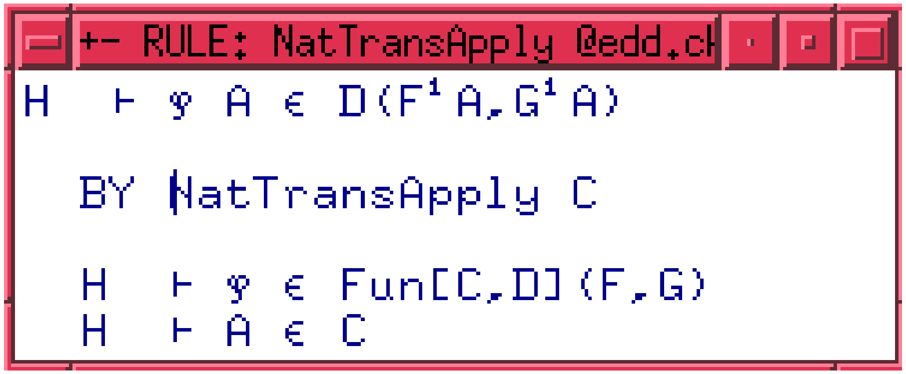
    E.g. `Comp{}(.`C`; .`g`; .`f`)` is represented as $g \circ f$

- **Represent inference rules as top-down rule objects**

  e.g.

$$\frac{\Gamma \vdash \varphi : \mathsf{Fun}[\mathsf{C},\mathsf{D}](F,G) \qquad \Gamma \vdash A : \mathsf{C}}{\Gamma \vdash \varphi\ A : \mathsf{D}(F^1 A, G^1 A)}$$

  is represented by



  Top-down rule needs category C as control parameter

  Judgments and equations are typed

  Nuprl’s rule compiler converts rule objects into rules that match first the line against the actual goal sequent and generate subgoals accordingly

- **Shallow embedding possible but not necessary**
  - Useful only for a validation of the implemented calculus rules

Automating proofs: standard techniques

Decomposition + extensionality + term rewriting

- **Structure of terms and types yields applicable rules**
  – A conclusion $\varphi\ A \in \mathsf{D}(X, Y)$ suggests using NatTransApply
  – Block application of analysis rules that create subgoals previously decomposed by a synthesis rule
- **Determine rule parameters of the rules via type checking**
  – The parameter C in NatTransApply must be the type of A
- **Prove equalities through rewriting**
  – Convert equalities into directed rewrite rules
  – Use Knuth-Bendix completion to make the rewrite system confluent
- **Eliminate redundant subgoals using rule wrappers**
  – Some rules generate similar subgoals in different proof branches
  – Controlled application of cut rule reduces proof size by 90%
  e.g. 3,000 proof steps for $\mathsf{Fun}\,[\mathsf{C}\times\mathsf{D}, \mathsf{E}] \simeq \mathsf{Fun}\,[\mathsf{C}, \mathsf{Fun}\,[\mathsf{D}, \mathsf{E}]\,]$ instead of 30,000

Automating reasoning specific to category theory

- **How do we prove $\mathsf{Fun}\,[\mathsf{C}{\times}\mathsf{D},\mathsf{E}] \simeq \mathsf{Fun}\,[\mathsf{C},\mathsf{Fun}\,[\mathsf{D},\mathsf{E}]\,]$?**
  - Proof requires specification of two (inverse) functors

    $\vartheta : \mathsf{Fun}\,[\mathsf{C}{\times}\mathsf{D},\mathsf{E}]{\rightarrow}\mathsf{Fun}\,[\mathsf{C},\mathsf{Fun}\,[\mathsf{D},\mathsf{E}]\,]$

    and $\eta : \mathsf{Fun}\,[\mathsf{C},\mathsf{Fun}\,[\mathsf{D},\mathsf{E}]\,]{\rightarrow}\mathsf{Fun}\,[\mathsf{C}{\times}\mathsf{D},\mathsf{E}]$
  - We know $\vartheta^1 f = \lambda A.\lambda B.f(A,B) : \mathsf{Fun}\,[\mathsf{C},\mathsf{Fun}\,[\mathsf{D},\mathsf{E}]\,]$ for $f{:}\mathsf{Fun}\,[\mathsf{C}{\times}\mathsf{D},\mathsf{E}]$
    but that is only the object component of the resulting functor
  - We also need its morphism component and the transformation ϑ^2
- **Specify functors component-wise**
  - First order specification via equations for all subcomponents of ϑ / η
  - E.g. use $\vartheta^1\mathtt{f}^1\mathtt{A}^1\mathtt{B} \;\equiv\; \mathtt{f}^1\mathtt{<A,B>}$ and $\vartheta^1\mathtt{f}^1\mathtt{A}^2\mathtt{g} \;\equiv\; \mathtt{f}^2\mathtt{<1}_A\mathtt{,g>}$
    instead of $\vartheta \;\equiv\; \lambda\mathtt{f,A...}$, which is no category theoretic expression
  - Only these equations will be used during the (first order) proof
  - Standard techniques can easily verify correctness of the functors

But how do we find these specifications?

Two questions that I had to ask

- **How can we determine the specification of ϑ and η?**

  DK: "*The only possible solution can be found by looking at the types*"

- **How can we prove that ϑ and η are natural in $\mathcal{C}, \mathcal{D}, \mathcal{E}$?**

  DK: "*Once the domain/codomain of ϑ and η can be derived from the construction of the two categories, the rest should be obvious*"

So, for the mathematician the solution is obvious

Can these ideas be automated?

Witness construction can in fact be automated

```
*- PRF : Iso-curry-v2 @edd.ck @sem

* top
∀C,D,E:Categories.  Fun[C×D,E] ≙ Fun[C,Fun[D,E]]

* BY ProveIso·
```

```
Iso-curry-v2 2012_01_25-AM-08_13_41 @edd.ck @sem

* top
∀C,D,E:Categories.  Fun[C×D,E] ≙ Fun[C,Fun[D,E]]

* BY UnravelStatement|

* 1

1. C : Categories
2. D : Categories
3. E : Categories
⊢ ∃ϑ:Fun[Fun[C×D,E],Fun[C,Fun[D,E]]].
 ∃η:Fun[Fun[C,Fun[D,E]],Fun[C×D,E]]. ϑ and η are inverse

* BY GuessFunctors

* 1 1

4. theta : Fun[Fun[C×D,E],Fun[C,Fun[D,E]]]
5. ϑ^1 F^1 A^1 X ≡ F^1 <A, X>
∧ ϑ^1 F^1 A^2 k ≡ F^2 <1A, k>
∧ ϑ^1 F^2 f X ≡ F^2 <f, 1X>
∧ ϑ^2 φ X X1 ≡ φ <X, X1>
6. eta : Fun[Fun[C,Fun[D,E]],Fun[C×D,E]]
7. η^1 F^1 <A, X> ≡ F^1 A^1 X
∧ η^1 F^2 <f, g> ≡ ((F^2 f cod(g))∘F^1 dom(f)^2 g)
∧ η^2 φ <A, X1> ≡ φ A X1
⊢ ϑ and η are inverse

* BY AutoCAT2
```

Developing a heuristic for witness construction

Find a functor $\vartheta : \mathsf{Fun}[\mathsf{C}\times\mathsf{D},\mathsf{E}] \to \mathsf{Fun}[\mathsf{C},\mathsf{Fun}[\mathsf{D},\mathsf{E}]]$

- **Create a specification from type information**
  - Generate typing conditions by applying decomposition rules
  - Construct the "simplest" term that satisfies these conditions
    - · Only the known parameters of the functor may be used
    - · Construction should be based on "obvious" ideas

- **Rule applications yield four conditions**

1.	$\vartheta^1 G \in \mathsf{Fun}[\mathsf{C},\mathsf{Fun}[\mathsf{D},\mathsf{E}]]$	for $G \in \mathsf{Fun}[\mathsf{C}\times\mathsf{D},\mathsf{E}]$
1.1	$(\vartheta^1 G)^1 A \in \mathsf{Fun}[\mathsf{D},\mathsf{E}]$	for $A \in \mathsf{C}$
1.1.1	$((\vartheta^1 G)^1 A)^1 X \in \mathsf{E}$	for $X \in \mathsf{D}$
1.1.2	$((\vartheta^1 G)^1 A)^2 h \in \mathsf{E}(((\vartheta^1 G)^1 A)^1 X, ((\vartheta^1 G)^1 A)^1 Y)$	for $h \in \mathsf{D}(X,Y)$
1.2.	$(\vartheta^1 G)^2 f \in \mathsf{Fun}[\mathsf{D},\mathsf{E}]((\vartheta^1 \mathsf{G})^1 \mathsf{A}, (\vartheta^1 \mathsf{G})^1 \mathsf{B})$	for $f \in \mathsf{C}(A,B)$
1.2.1.	$((\vartheta^1 G)^2 f) X \in \mathsf{E}(((\vartheta^1 G)^1 A)^1 X, ((\vartheta^1 G)^1 B)^1 X)$	for $X \in \mathsf{D}$
2.	$\vartheta^2 \varphi \in \mathsf{Fun}[\mathsf{C},\mathsf{Fun}[\mathsf{D},\mathsf{E}]](\vartheta^1 \mathsf{F}, \vartheta^1 \mathsf{G})$	for $\varphi \in \mathsf{Fun}[\mathsf{C}\times\mathsf{D},\mathsf{E}](\mathsf{F},\mathsf{G})$

$\vdots$

WITNESS CONSTRUCTION: SOLVING CONDITIONS I

- **Construct specifications for** $((\vartheta^1 G)^1 A)^1 X \in \mathsf{E}$
  - Available information: $G \in \mathsf{Fun}\,[\mathsf{C} \times \mathsf{D}, \mathsf{E}]$, $A \in \mathsf{C}$, $X \in \mathsf{D}$
- **There is only one meaningful solution**
  - To find an object in E apply G^1 to some $z \in \mathsf{C} \times \mathsf{D}$
  - Objects in $\mathsf{C} \times \mathsf{D}$ are pairs of objects $x \in \mathsf{C}$ and $y \in \mathsf{D}$
  - Only known object in C is A

    Only known object in D is X
  - z must be the pair $\langle A, X \rangle$
  - Construct subspecification $((\vartheta^1 G)^1 A)^1 X \;=\; G^1 \langle A, X \rangle$

Witness construction: solving conditions II

- **Solve** $((\vartheta^1 G)^2 f)X \in \mathsf{E}(((\vartheta^1 G)^1 A)^1 X, ((\vartheta^1 G)^1 B)^1 X)$
  - Available information: $G \in \mathsf{Fun}[\mathsf{C}\times\mathsf{D}, \mathsf{E}]$, $f \in \mathsf{C}(A, B)$, $X \in \mathsf{D}$
  - Also known from the previous step: $((\vartheta^1 G)^1 A)^1 X = G^1\langle A, X\rangle$
  - Thus to construct: $((\vartheta^1 G)^2 f)X \in \mathsf{E}(G^1\langle A, X\rangle, G^1\langle B, X\rangle)$
- **There is only one meaningful solution**
  - To find a morphism in $\mathsf{E}(G^1\langle A, X\rangle, G^1\langle B, X\rangle$ apply G^2 to some $k \in \mathsf{C}\times\mathsf{D}(\langle A, X\rangle, \langle B, X\rangle)$
  - Morphisms in $\mathsf{C}\times\mathsf{D}(\langle A, X\rangle, \langle B, X\rangle)$ are pairs of morphisms $g \in \mathsf{C}(A, B)$ and $h \in \mathsf{D}(X, X)$
  - Only known morphism in $\mathsf{C}(A, B)$ is f
    Only known morphism in $\mathsf{D}(X, X)$ is 1_X
  - Construct subspecification $((\vartheta^1 G)^2 f)X = G^2\langle f, 1_X\rangle$

Intuitively clear – how to automate?

A calculus for witness construction

Construct specifications from typing conditions

- **Formulate construction requirements as rules**
  - E.g.: to use $F \in \mathsf{Fun}[\mathsf{C},\mathsf{D}]$ when constructing some $x \in \Delta$, construct some $z \in \mathsf{C}$ and use $y = F^1 z \in \mathsf{D}$ for the remaining construction of x

$$\Gamma, F : \mathsf{Fun}[\mathsf{C},\mathsf{D}] \vdash x \in \Delta \qquad \textsf{specs } EQ_1 \cup EQ_2[F^1 z/y]$$

$$\Gamma \vdash z \in \mathsf{C} \qquad \textsf{specs } EQ_1$$

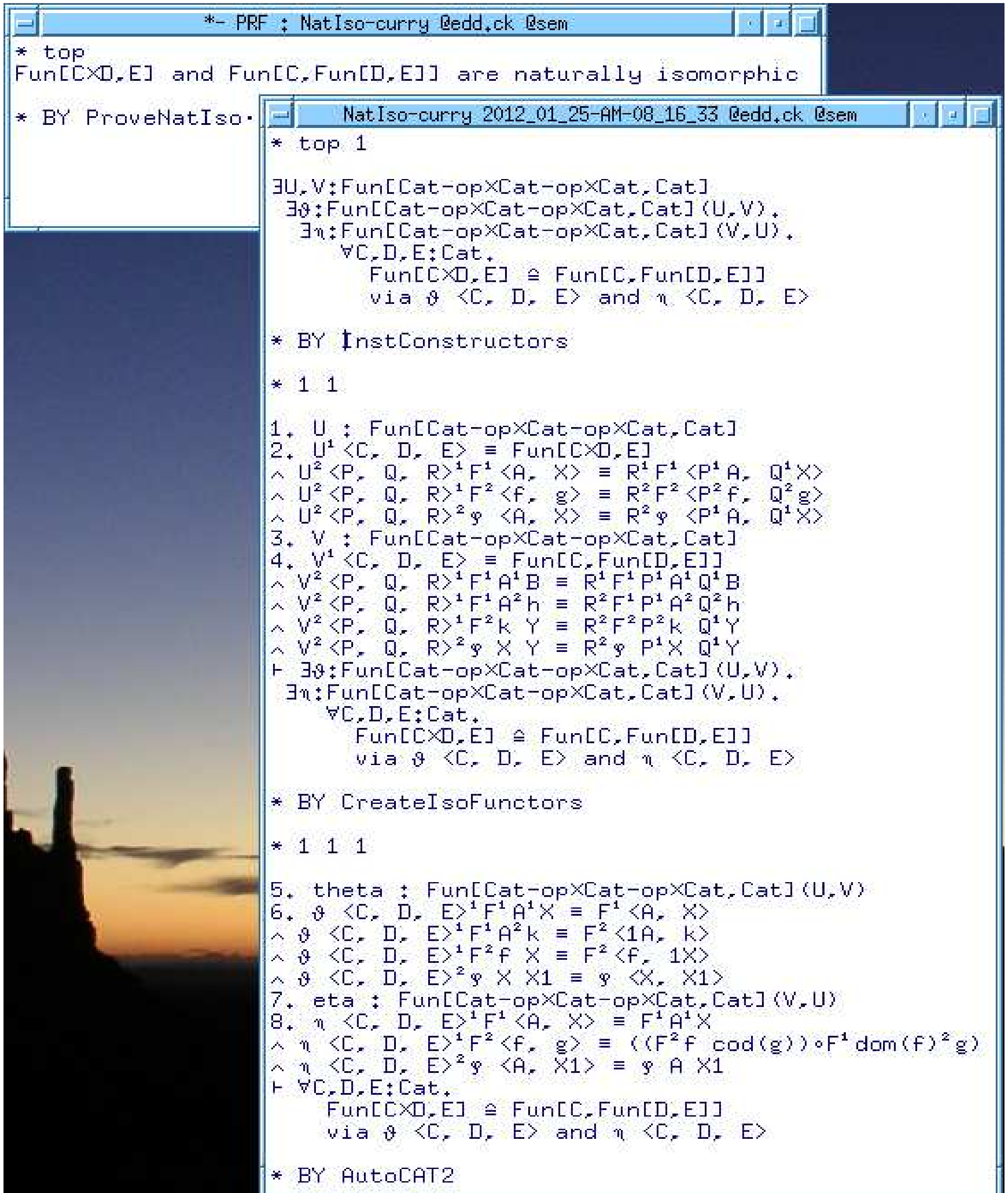
$$\Gamma, y : \mathsf{D} \vdash x \in \Delta \qquad \textsf{specs } EQ_2$$

  - Compose and reduce specification equations of all the subgoals

- **Actual construction of witnesses happens at the leaf level**
  - Hypothesis: $\Gamma, z{:}\Delta[V_1, ..V_n] \vdash x \in \Delta[T_1, ..T_n]$ specs $\{x{=}z, V_1{=}T_1, .., V_n{=}T_n\}$
  - Identity: $\Gamma, A{:}\mathsf{C} \vdash f \in \mathsf{C}(A, A)$ specs $\{f{=}1_A\}$

- **Implementation:**
  - Apply applicable rules in the order of "simplicity"

How to prove naturality?



- **Determine** (co-)domain U, V of ϑ and η
- **Specify ϑ and η**
  - Witness construction $\checkmark$
- **Verify** type of U, V, ϑ, η; duality of ϑ, η
  - `AutoCAT` $\checkmark$

A calculus for the construction of (co-)domains

- **U, V are 'constructor functions' on** Cat
  - U^1, V^1 construct the two isomorphic categories
    - e.g. $U^1(\mathsf{C}, \mathsf{D}, \mathsf{E}) = \mathsf{Fun}\,[\mathsf{C} \times \mathsf{D}, \mathsf{E}]$ and $V^1(\mathsf{C}, \mathsf{D}, \mathsf{E}) = \mathsf{Fun}\,[\mathsf{C}, \mathsf{Fun}\,[\mathsf{D}, \mathsf{E}]\,]$
  - U^2, V^2 construct functors on these categories
    - e.g.. $U^2(f, g, h) \in \mathsf{Fun}\,[\mathsf{Fun}\,[\mathsf{C} \times \mathsf{D}, \mathsf{E}]\,, \mathsf{Fun}\,[\mathsf{C}' \times \mathsf{D}', \mathsf{E}']\,]$
  - U, V are composed from simple constructors: e.g. $U = \mathcal{F}_{\mathsf{Fun}} \circ (\mathcal{F}_{\mathsf{Prod}}, \mathcal{F}^3_{\mathsf{proj3}})$
- **Specify basic constructor functions and projections**
  - For product-, functor-, and dual categories
  - e.g.. $\mathcal{F}_{\mathsf{Fun}}{}^1(\mathsf{C},\mathsf{D}) = \mathsf{Fun}\,[\mathsf{C}, \mathsf{D}]$ for $\mathsf{C} \in \mathsf{Cat}^{\mathsf{op}}, \mathsf{D} \in \mathsf{Cat}$

    $\mathcal{F}_{\mathsf{Fun}}{}^2(h_1,h_2)^1(F) = h_2 \circ F \circ h_1$ and $\mathcal{F}_{\mathsf{Fun}}{}^2(h_1,h_2)^2(\varphi) = h_2{}^2 \circ \varphi \circ h_1{}^1$

    for $(h_1,h_2) \in \mathsf{Cat}^{\mathsf{op}} \times \mathsf{Cat}((C, D),\ (C', D')),\ F \in Fun[\mathcal{C}, \mathcal{D}],\ \varphi \in Fun[\mathsf{C}, \mathsf{D}](F, G)$
  - Yields 'most simple functor' that satisfies the typing conditions

The specification of a composed constructor is determined by composing and reducing the corresponding equations

Results and insights

- **Implementation of calculus for reasoning about category**
  - Abstractions and display forms crucial for comprehensibility
  - Rule objects and rule compiler essential for faithful implementation
  - Tactic mechanism supports automation of reasoning patterns
  - Nested abstraction levels in proof objects make proofs comprehensible
- **Proofs of (natural) isomorphisms completely automated**
  - $\mathsf{Fun}\,[\mathsf{C}{\times}\mathsf{D},\mathsf{E}] \simeq \mathsf{Fun}\,[\mathsf{C},\mathsf{Fun}\,[\mathsf{D},\mathsf{E}]]$, $\mathsf{C}{\times}\mathsf{D} \simeq \mathsf{D}{\times}\mathsf{C}$, $(\mathsf{C}^{op})^{op} \simeq \mathsf{C}$, ...
- **Elementary category theory well-suited for automation.**
  - Formal proofs have thousands of basic inferences
  - Most proof steps are driven by typing considerations
  - Witness construction follows standard patterns of reasoning
- **Intellectualy trivial insights have in fact trivial proofs**
  - Computers can find them without using sophisticated heuristics

Where can we go from here?

- **Automate more of elementary category theory**
  - Use calculus for witness construction to find simple functors
  - Use calculus of constructor functions to find natural transformations
  - Can we formalize the rusults on Brzozowski's Algorithm?
- **Introduce higher-level reasoning steps**
  - Can we use compositional reasoning based on theorems?
    - e.g. if $\mathsf{C} \simeq \mathsf{D}$ and $\mathsf{E}[X] \simeq \mathsf{E}'[X]$ can we prove $\mathsf{E}[\mathsf{C}] \simeq \mathsf{E}'[\mathsf{D}]$?
- **Can we extract evidence for naturality from proofs?**
  - E.g. naturality of an isomorphism between two categories?
  - Should be possible for all categories that have a term representation (i.e. can be described using constructor functors)
  - Inductive construction seems obvious – can we prove that formally?

Automated Fast-Track Reconfiguration of Group Communication Systems

Christoph Kreitz

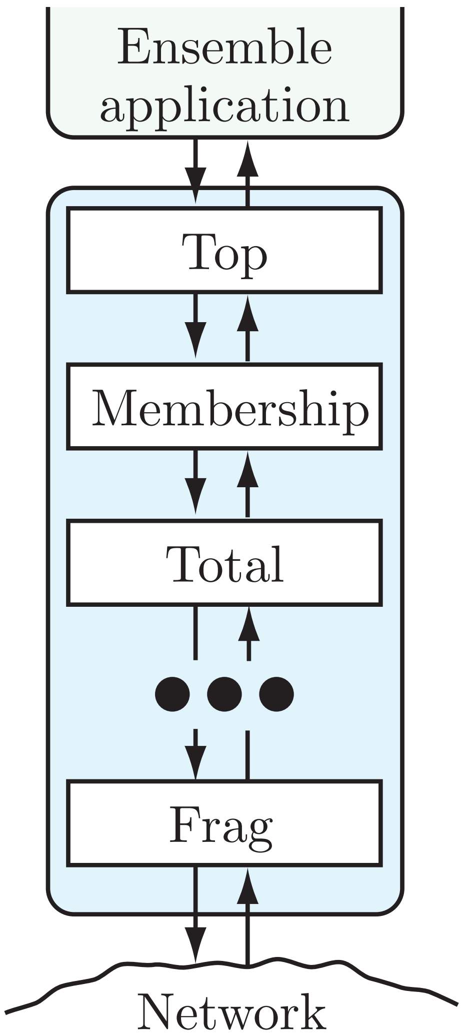
Department of Computer Science, Cornell University
&
Institut für Informatik, Universität Potsdam

1. Group Communication in Ensemble
2. Embedding Ensemble's code into Nuprl
3. Formal optimization of protocol stacks
4. Lessons learned

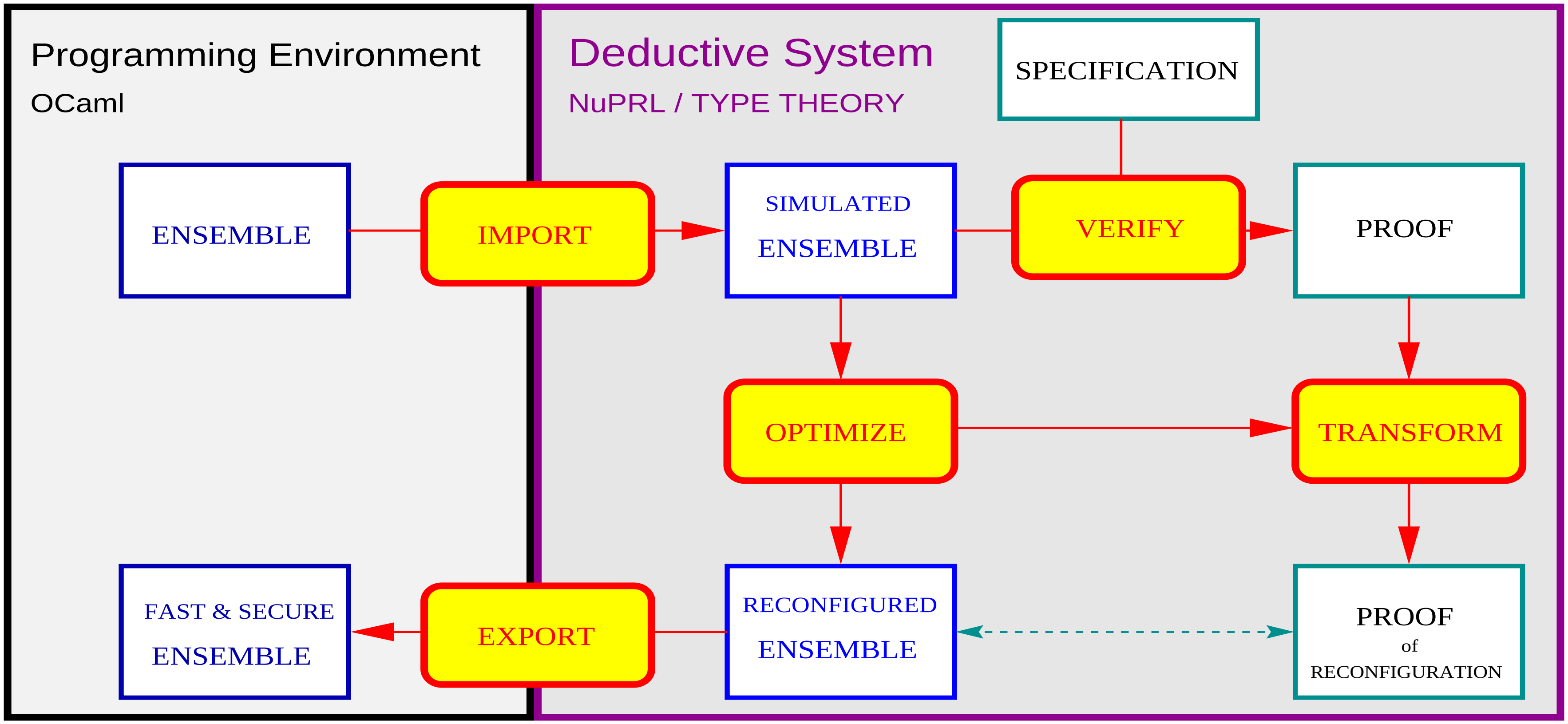
Joint work with Jason Hickey, Mark Hayden, Bob Constable, and Robbert VanRenesse

The Ensemble Group Communication Toolkit

- **Modular group communication system**
  - Developed by Cornell's System Group (Ken Birman)
  - Used commercially (BBN, JPL, Segasoft, Alier, Nortel Networks)
- **Architecture: stack of micro-protocols**
  - Select from more than 60 micro-protocols for specific tasks
  - Modules can be stacked arbitrarily
    System can easily be tailored to specific applications
  - Modeled as state/event machines
- **Implementation in** Objective Caml (INRIA)
  - Easy maintenance (small code, good data structures)
  - Mathematical semantics, strict data type concepts
  - Efficient compilers and type checkers



FORMAL REASONING ABOUT A REAL-WORLD SYSTEM

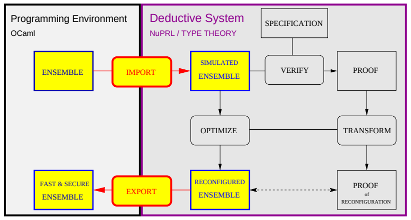


Link the ENSEMBLE **and** Nuprl **systems**

– Embed ENSEMBLE's code into Nuprl's language

– Verify protocol components and system configurations

– Optimize performance of configured systems



EMBEDDING ENSEMBLE'S CODE INTO NUPRL

- **Develop type-theoretical semantics of OCaml**
  - Functional core, pattern matching, exceptions, references, modules,…
- **Implement using Nuprl's definition mechanism**
  - Represent OCaml's semantics via abstraction objects
  - Represent OCaml's syntax using associated display objects
- **Develop programming logic for OCaml**
  - Implement as rules derived from the abstract representation
  - Raises the level of formal reasoning from Type Theory to OCaml
- **Develop tools for importing and exporting code**
  - Translators between OCaml program text and Nuprl terms

OCaml Semantics: The functional core

- **Basic OCaml expressions similar to CTT terms**
  - Numbers, tuples, lists etc. can be mapped directly onto CTT terms
- **Complex data structures have to be simulated**

  Records $\{f_1 = e_1; \ldots; f_n = e_n\}$ are functions in `f:FIELDS`$\to T$`[f]`
  - Abstraction for representing the semantics of record expressions

    `RecordExpr(`*field*`;`*e*`;`*next*`)` $\equiv$ λ`f. if f=`*field* `then` *e* `else` *next*`(f)`
  - Display form for representing the flexible syntax of record expressions

	`{`*field*`=`*e*`;` *next*`}`	$\equiv$	`RecordExpr(`*field*`;`*e*`;`*next*`)`
	`{`*field*`=`*e*`}`	$\equiv$	`RecordExpr(`*field*`;`*e*`;`λ`f.())`
`HD::`	`{`*field*`=`*e*`; #`	$\equiv$	`RecordExpr(`*field*`;`*e*`;#)`
`TL::`	*field*`=`*e*`; #`	$\equiv$	`RecordExpr(`*field*`;`*e*`;#)`
`TL::`	*field*`=`*e*`}`	$\equiv$	`RecordExpr(`*field*`;`*e*`;`λ`f.())`

- **Sufficient for representing micro protocols**
  - Simple state-event machines, encoded via updates to certain records
  - Transport module and protocol composition require imperative model

Extensions of the semantical model (1)

- **Type Theory is purely functional**
  - Terms are evaluated solely by reduction
  - OCaml has pattern matching, reference cells, exceptions, modules, …
- **Modelling Pattern Matching:** `let` $\boldsymbol{pat}$`=`$\boldsymbol{e}$ `in` $\boldsymbol{t}$

  "Variables of pat in t are bound to corresponding values of e"
  - Evaluation of OCaml-expressions uses an environment of bindings
  - Patterns are functions that modify the environment of expressions

$$\begin{array}{lcl} \boldsymbol{x} & \equiv & \lambda\texttt{val,t.}\lambda\texttt{env. t (env@}\{\boldsymbol{x}\mapsto\texttt{val}\}\texttt{)} \\ \boldsymbol{p}_1\texttt{,}\boldsymbol{p}_2 & \equiv & \lambda\texttt{val,t.}\lambda\texttt{env. let <v}_1\texttt{,v}_2\texttt{>=val in (}\boldsymbol{p}_1\texttt{ v}_1\texttt{ (}\boldsymbol{p}_2\texttt{ v}_2\texttt{ t)) env} \\ \{f_1\texttt{=}p_1;\,..\,;f_n\texttt{=}p_n\} & \equiv & \lambda\texttt{val,t.}\lambda\texttt{env. } p_1\texttt{ (val } f_1\texttt{) (..(}p_n\texttt{ (val } f_n\texttt{) t)..) env} \\ \vdots & & \vdots \end{array}$$

  - Local bindings are represented as applications of these functions

$$\texttt{let } \boldsymbol{p}\texttt{=}\boldsymbol{e}\texttt{ in } \boldsymbol{t} \;\equiv\; \lambda\texttt{env. (}\boldsymbol{p}\texttt{ (}\boldsymbol{e}\texttt{ env) }\boldsymbol{t}\texttt{) env}$$

Extensions of the semantical model (2)

- **Modelling Reference cells**
  - Evaluation of OCaml-expressions may lookup/modify a global store
  - The global store is represented as table with addresses and values

```
ref(e)   ≡ λs,env. let <v,s1>=e s env in
                     let addr=NEW(s1) in <addr, s1[addr←v]>
!e       ≡ λs,env. let <addr,s1>=e s env in <s1[addr], s1>
e1:=e2   ≡ λs,env. let <v,s1>=e2 s env in
                     let <addr,s2>=e1 s1 env in <(), s2[addr←v]>
```

- **Modelling Exceptions**
  - Expressions like `x/y` may raise exceptions, which can be caught
  - Exceptions must have the same type as the expression that raises them
  - An OCaml type T must be represented as `EXCEPTION` + T

- **Modules**
  - Modules are second class objects that structure the name space
  - Modules are represented by operations on a global environment

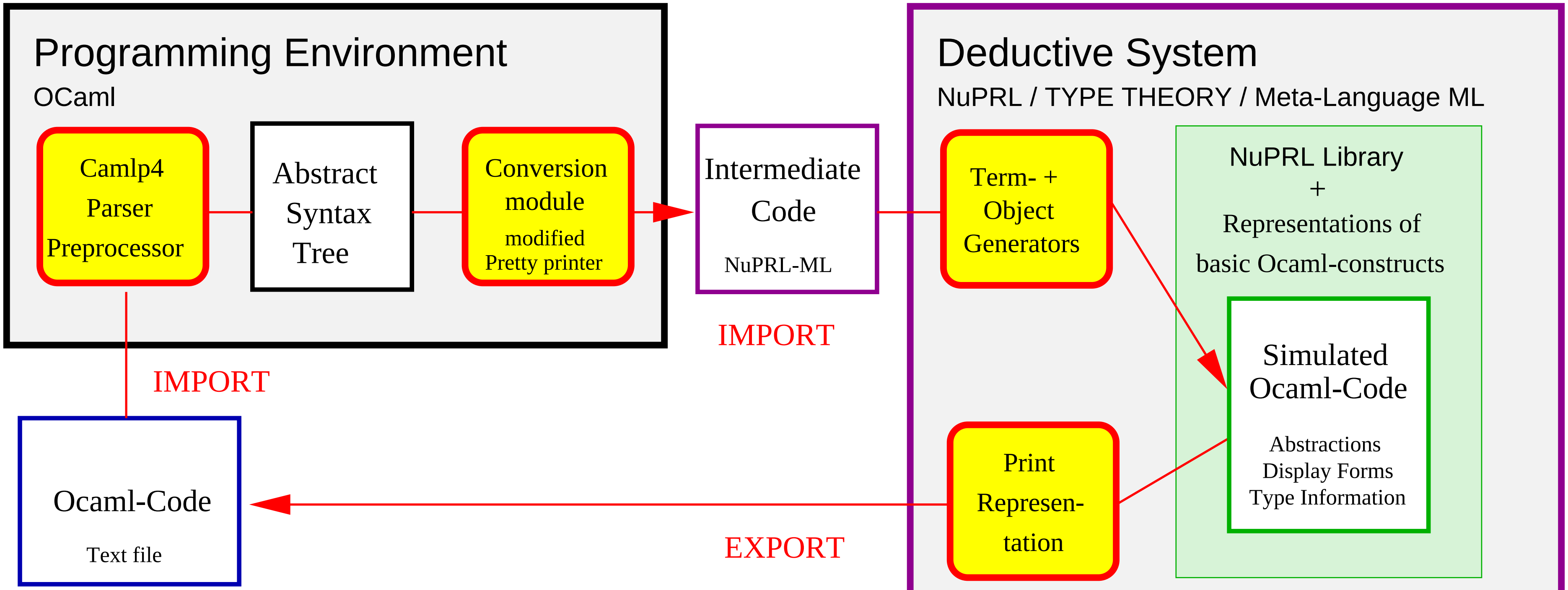
Summary of the formal model

- **OCaml expressions are represented as functions**
  - Evaluation depends on environment and store
  - Evaluation results in value or exception and an updated store
  - Nuprl type is $\texttt{STORE} \rightarrow \texttt{ENV} \rightarrow (\texttt{EXCEPTION} + T) \times \texttt{STORE}$
- **Equivalent to Wright/Felleisen model**
  - The standard model for building ML compilers
  - Model combines several mechanisms for evaluating ML programs
  - Nuprl representation simulates these models functionally

$\Downarrow$

Genuine OCaml code may occur in formal theorems

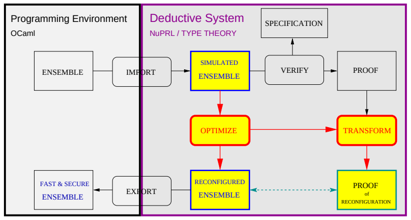
Importing and Exporting System Code



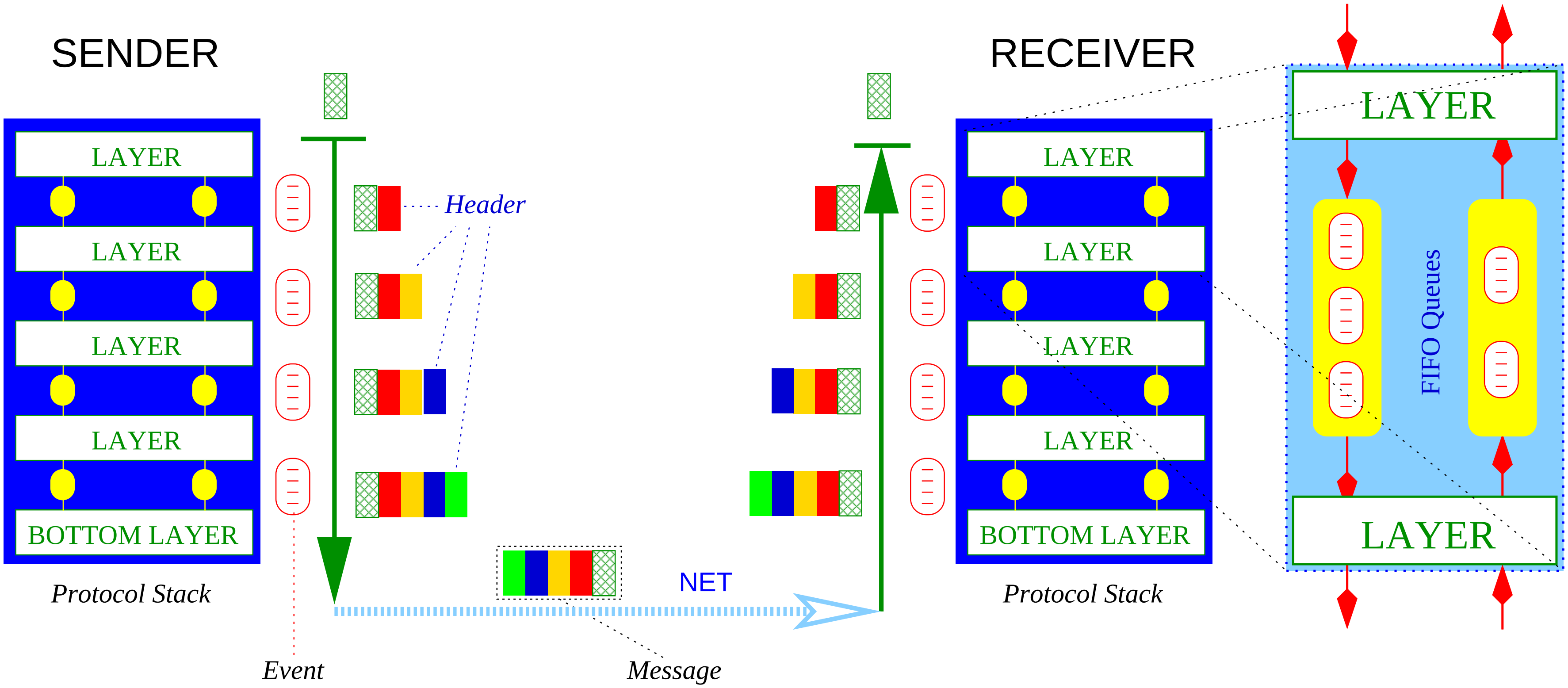
Import: – Parse with Camlp4 parser-preprocessor
– Convert abstract syntax tree into term- & object generators
– Generators perform second pass and create Nuprl library objects

Export: – Print-representation is genuine OCaml-code

Makes actual ENSEMBLE code available for formal reasoning



Optimization of Protocol Stacks



Performance loss: redundancies, internal communication, large message headers

Optimizations: bypass-code for common execution sequences, header compression

Need formal methods to do this correctly

Example Protocol Stack Bottom::Mnak::Pt2pt

Trace downgoing Send events and upgoing Cast events

Bottom (200 lines)

```
let name = Trace.source_file "BOTTOM"

type header = NoHdr | ... | ...

type state = {mutable all_alive : bool ; ... }

let  init _ (ls,vs) = {.........}

let  hdlrs s (ls,vs)
      {up_out=up;upnm_out=upnm;
       dn_out=dn;dnlm_out=dnlm;dnnm_out=dnnm}
 = ...
 let up_hdlr ev abv hdr =
  match getType ev, hdr with
  | (ECast|ESend), NoHdr ->
    if s.all_alive  or not (s_bottom.failed.(getPeer ev))
       then up ev abv
       else free name ev
  | :
 and uplm_hdlr ev hdr    = ...
 and upnm_hdlr ev        = ...
 and dn_hdlr ev abv      =
  if s.enabled then
    match getType ev with
    | ECast          -> dn ev abv NoHdr
    | ESend          -> dn ev abv NoHdr
    | ECastUnrel     -> dn (set name ev[Type ECast]) abv Unrel
    | ESendUnrel     -> dn (set name ev[Type ESend]) abv Unrel
    | EMergeRequest -> dn ev abv MergeRequest
    | EMergeGranted -> dn ev abv MergeGranted
    | EMergeDenied  -> dn ev abv MergeDenied
    | _ -> failwith "bad down event[1]"
   else (free name ev)
 and dnnm_hdlr ev        = ...
 in {up_in=up_hdlr;uplm_in=uplm_hdlr;upnm_in=upnm_hdlr;
     dn_in=dn_hdlr;dnnm_in=dnnm_hdlr}

let l args vs = Layer.hdr init hdlrs args vs

Layer.install name (Layer.init l)
```

Mnak (350 lines)

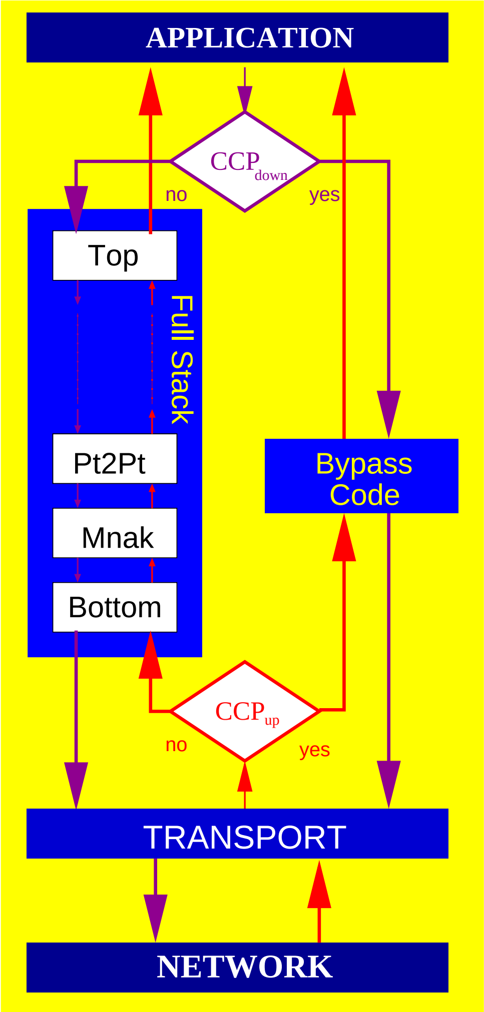
```
let init ack_rate (ls,vs) = {.........}
let hdlrs s (ls,vs) { ......... }
 = ...
 let ...
 and dn_hdlr ev abv      =
   match getType ev with
   | ECast ->
      let iov = getIov ev in
      let buf = Arraye.get s.buf ls.rank in
      let seqno = Iq.hi buf in
      assert (Iq.opt_insert_check buf seqno) ;
      Arraye.set s.buf ls.rank
             (Iq.opt_insert_doread buf seqno iov abv) ;
      s.acct_size <- s.acct_size + getIovLen ev ;
      dn ev abv (Data seqno)
   | _ -> dn ev abv NoHdr
    :
```

Pt2pt (250 lines)

```
let init _ (ls,vs) = {.........}
let hdlrs s (ls,vs) { ......... }
 = ...
 let ...
 and dn_hdlr ev abv      =
  match getType ev with
  | ESend ->
     let dest = getPeer ev in
     if dest = ls.rank then (
       eprintf "PT2PT:%s\nPT2PT:%s\n"
        (Event.to_string ev) (View.string_of_full (ls,vs));
       failwith "send to myself" ;
     ) ;
     let sends = Arraye.get s.sends dest in
     let seqno = Iq.hi sends in
     let iov = getIov ev in
     Arraye.set s.sends dest (Iq.add sends iov abv) ;
     dn ev abv (Data seqno)
  | _ -> dn ev abv NoHdr
   :
```

Fast-path Optimization with Nuprl

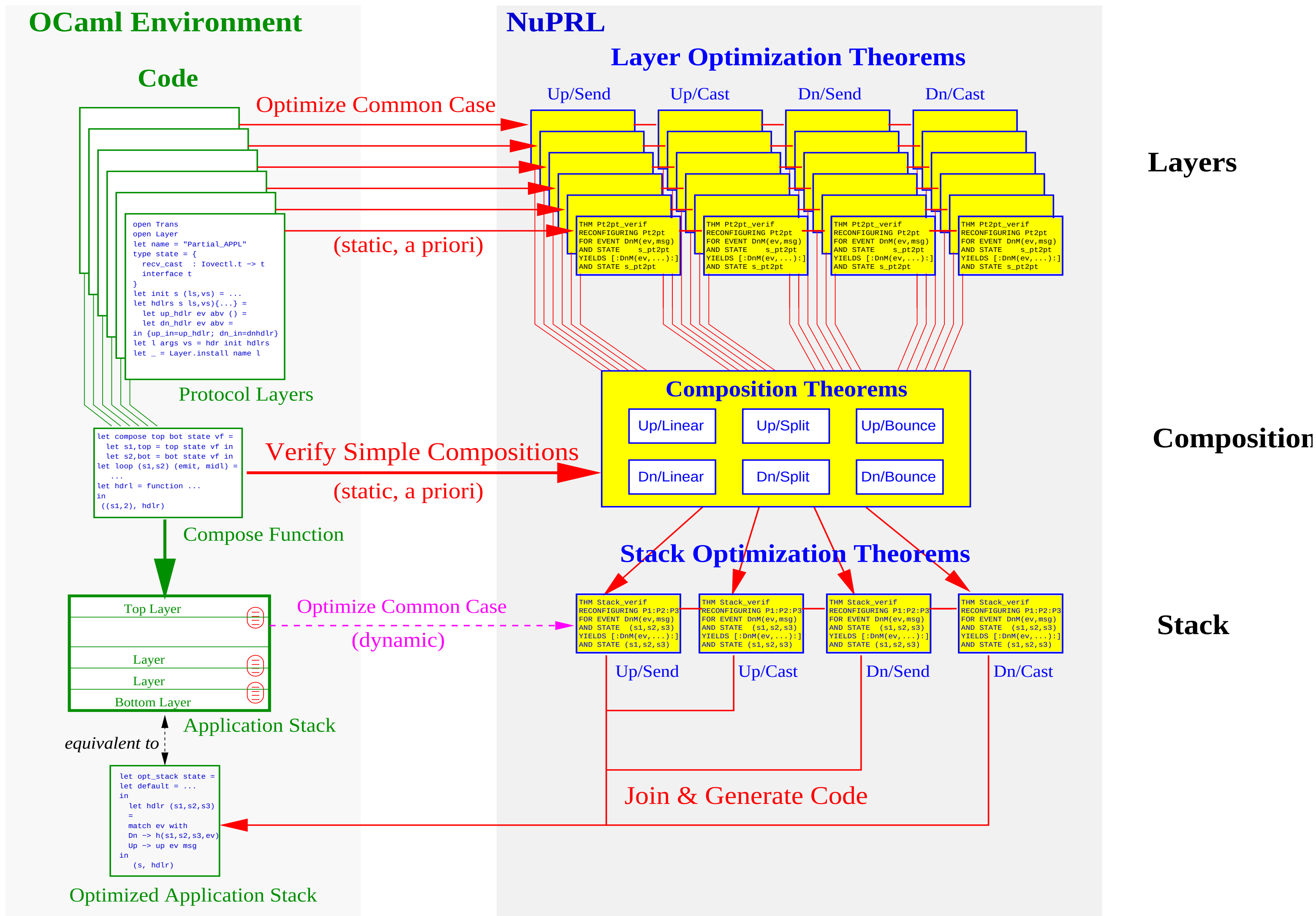
- **Identify Common Case**
  - Events and protocol states of regular communication
  - Formalize as **C**ommon **C**ase **P**redicate
- **Analyze path of events through stack**
- **Isolate code for fast-path**
- **Integrate code for compressing headers of common messages**
- **Generate bypass-code**
  - Insert CCP as runtime switch



Methodology: compose formal optimization theorems

Fast, error-free, independent of programming language, **speedup factor 3-10**

Methodology: Compose Optimization Theorems



1. Use known optimizations of micro-protocols — A priori: ENSEMBLE + Nuprl experts
2. Compose into optimizations of protocol stacks — automatic: application designer
3. Integrate message header compression — automatic: ⋮
4. Generate code from optimization theorems and reconfigure system — automatic: ⋮

Static optimization of micro protocols

- **A-priori analysis of common execution sequences**
  - Generate local CCP from conditionals in a layer's code
- **Assuming the CCP, apply code transformations**
  - Controlled function inlining and symbolic evaluation (rewrite tactics)
  - Directed equality substitutions (lemma application)
  - Context-dependent simplifications (substitute part of CCP and rewrite)
- **Store result in library as optimization theorem**

```
OPTIMIZING LAYER Pt2pt
       FOR EVENT DnM (ev, msg)
       AND STATE s_pt2pt
        ASSUMING (getType ev) = ESend ∧ not (getPeer ev = ls.rank)
  YIELDS HANDLERS dn ev (Full (Data (Iq.hi
                     (Arraye.get s_pt2pt.sends (getPeer ev))), msg))
     AND UPDATES Iq.add (Arraye.get s_pt2pt.sends (getPeer ev))
                     (getIov ev) msg
```

  - Theorem proves correctness of the local optimization
  - Optimizations of micro protocols part of Ensemble's distribution

Dynamic Optimization of Application Stacks

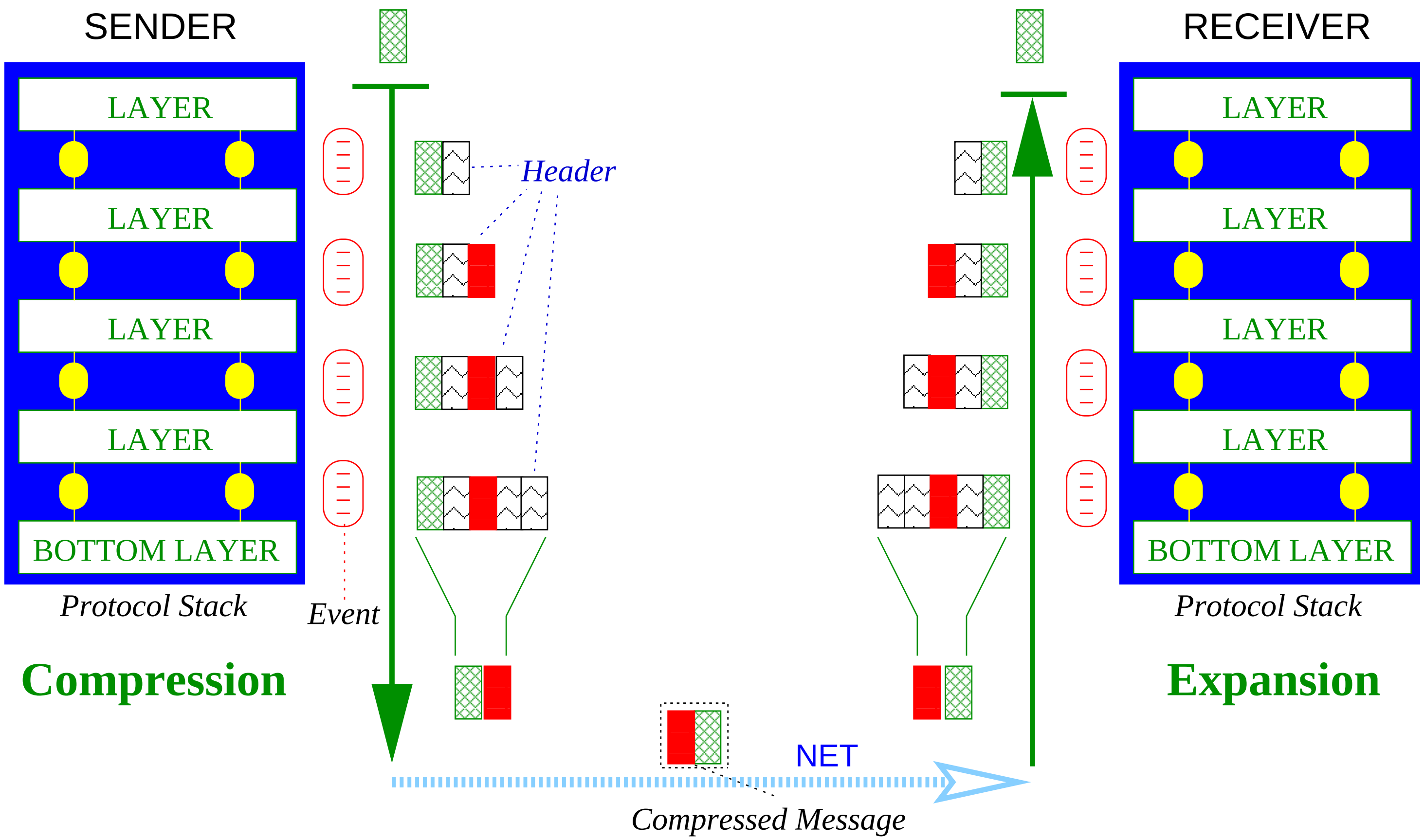
- **Compose Optimization Theorems**
  - Consult optimization theorems for individual layers
  - Apply composition theorems to generate stack optimization theorems (Linear, simple split, bouncing – send/receive)

```
  OPTIMIZING LAYER Upper
         FOR EVENT DnM(ev, hdr) AND STATE s_up
   YIELDS HANDLERS dn ev msg1 AND UPDATES stmt1
∧ OPTIMIZING LAYER Lower
         FOR EVENT DnM(ev, hdr1) AND STATE s_low
   YIELDS HANDLERS dn ev msg2 AND UPDATES stmt2

⇒ OPTIMIZING LAYER Upper || Lower
         FOR EVENT DnM(ev, hdr) AND STATE (s_up,s_low)
   YIELDS HANDLERS dn ev msg2 AND UPDATES stmt2; stmt1
```

  - Formal proof complex because of complex code for composition
- **Optimization of Protocol Stacks in Linear Time**
  - Use of optimization theorems reduces proof burden for optimizer
  - Pushbutton Technology: requires only configuration of stack

Header compression for fast-path code



Integrate compression into optimization process

– Generate code for compression and expansion from fast-path headers

– Combine optimization theorem for stack with compression theorems

– Optimized stack uses compressed headers directly

Example Optimization of `Bottom::Mnak::Pt2pt`

- **Generated optimization theorem for application stack**

```
OPTIMIZING LAYER Pt2pt::Mnak::Bottom
    FOR EVENT    DnM(ev, msg)
    AND STATE    (s_pt2pt, s_mnak, s_bottom)
    ASSUMING     getType ev = ESend ∧ getPeer ev ≠ ls.rank ∧ s_bottom.enabled
 YIELDS HANDLERS dn ev (Full(NoHdr, Full(NoHdr,
                          Full(Data(Iq.hi s_pt2pt.sends.(getPeer ev)),msg))))
    AND UPDATES  Iq.add (Arraye.get s_pt2pt.sends (getPeer ev))(getIov ev) msg
```

- **Generated code for header compression**

```
let compress hdr = match hdr with
   Full(NoHdr, Full(NoHdr, Full(Data(seqno), hdr))) -> OptSend(seqno, hdr)
 | Full(NoHdr, Full(Data(seqno), Full(NoHdr, hdr))) -> OptCast(seqno, hdr)
 | hdr                                              -> Normal(hdr)
```

- **Optimization theorem including header compression**

```
OPTIMIZING LAYER Pt2pt::Mnak::Bottom
    FOR EVENT    DnM(ev, msg)
    AND STATE    (s_pt2pt, s_mnak, s_bottom)
    ASSUMING     getType ev = ESend ∧ getPeer ev ≠ ls.rank ∧ s_bottom.enabled
 YIELDS HANDLERS dn ev (OptSend(Iq.hi s_pt2pt.sends.(getPeer ev), msg))
    AND UPDATES  Iq.add (Arraye.get s_pt2pt.sends (getPeer ev))(getIov ev) msg
```

Code generation

1. Convert Theorems into Code Pieces

– handlers + updates ↦ command sequence

– CCP ↦ conditional / case-expression

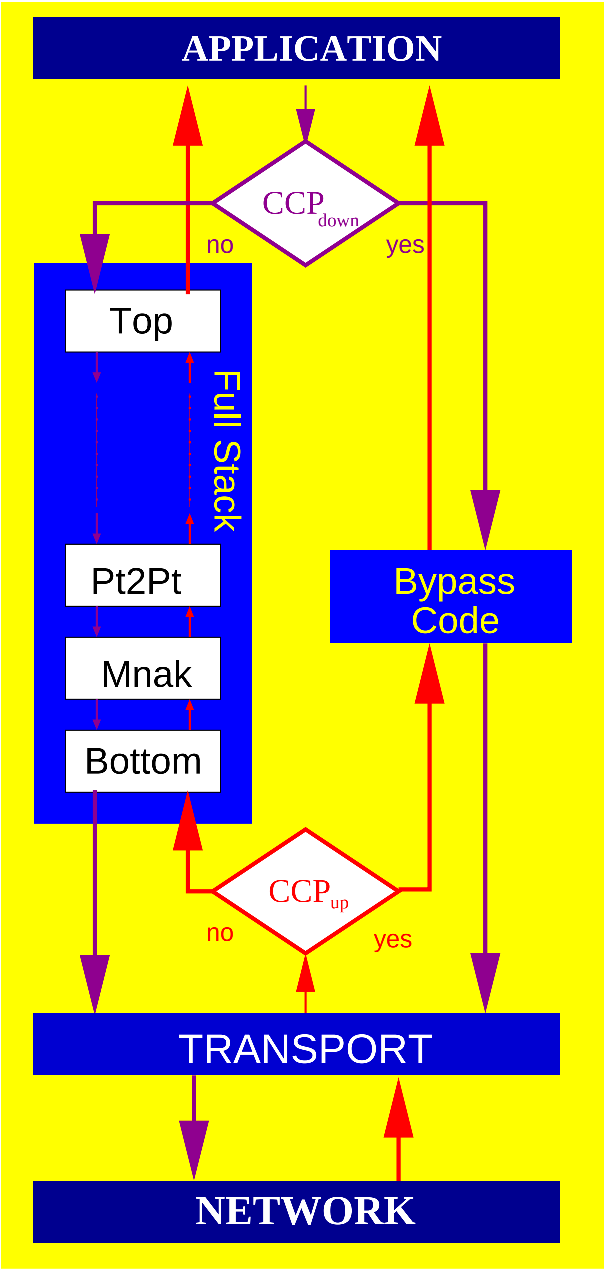
– Call original event handler if CCP fails

2. Assemble Code Pieces

– Case expression for 4 common cases (send/receive, broadcast/point-to-point)

– Call original event handler in final case

3. Export into OCaml environment



Fully automated,
generated code 3–10 times faster

Lessons learned

- **Results**
  - Type theory expressive enough to formalize today's software systems
  - Nuprl capable of supporting real design at reasonable pace
  - Formal optimization can significantly improve practical performance
  - Formal verification reveals errors even in well-investigated designs
- **Ingredients for success . . .**
  - Collaboration between systems and formal reasoning groups
  - Small and simple components, well-defined module composition
  - Implementation language with precise semantics
  - Formal models of: communication, programming language
  - Knowledge-based approach based on algorithmic knowledge
  - Cooperating reasoning tools

Future Challenges

The Ensemble case study is just a 'proof of concept'

- **We still need**
  – Better reasoning tools (w.r.t performance and application range)
  – Extensive library of formal algorithmic knowledge
  – More insights from increasingly complex applications
  – Tools for **synthesis** instead of just verification and optimization
  – Strong cooperation between research groups

  ... and most of all ... **more people to get involved**